An Incremental Programming Environment

RAUL MEDINA-MORA, STUDENT MEMBER, IEEE, AND PETER H. FEILER

Abstract—This paper describes an incremental programming environment (IPE) based on compilation technology, but providing facilities traditionally found only in interpretive systems. IPE provides a comfortable environment for a single programmer working on a single program.

In IPE the programmer has a uniform view of the program in terms of the programming language. The program is manipulated through a syntax-directed editor and its execution is controlled by a debugging facility, which is integrated with the editor. Other tools of the traditional tools cycle (translator, linker, loader) are applied automatically and are not visible to the programmer. The only interface to the programmer is the user interface of the editor.

Index Terms—Ada environments, incremental compilation, incremental program construction, interactive debugging, programming environments, programming methodology, software development environments, syntax-directed editing.

I. Introduction

A PROGRAMMING ENVIRONMENT supports programmers in the process of transforming specifications into working programs. This process involves creation and modification of programs, checking their consistency, generation of an executable form, and monitoring of the actual program behavior. Traditionally, a programming environment for compiled languages consisted of a high-level language and some simple tools. One of the main purposes of a programming environment is to make it easier to write more sophisticated programming systems. Programming languages contribute to this goal by supporting concepts such as modularization and data abstraction. The Ada language is a product of the evolution of these concepts. Some improvements have also been made to the different tools of the environment in isolation, but environments still consist of a set of independent tools. In programming environments for interpreted languages emphasis has been placed on the functionality of tools and their integration. Such an environment provides the interactive behavior and flexibility that is desired for experimental program development. Interlisp [1] is an example of such an environment.

Manuscript received August 10, 1980; revised March 9, 1981. This work was supported in part by the Software Engineering Division of CENTACS/CORADCOM, Fort Monmouth, NJ, and the National University of Mexico, Mexico City, Mexico. This is a revised version of a paper presented at the Fifth International Conference on Software Engineering.

R. Medina-Mora is with the Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213, on leave from the Department of Computer Science, Institute of Applied Mathematics and Systems, National University of Mexico, Mexico City, Mexico.

P. H. Feiler is with the Siemens Corporation with residence at the Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

In IPE (*incremental programming environment*) we attempt to provide the comfort of a flexible and interactive programming environment for compiler-based languages, i.e., to support development and maintenance of longer lived programs. This is achieved by integration of the software tools into one common goal. A common program representation provides the means of communication among the tools. The environment has knowledge about the objects it manipulates and their current state. It is, therefore, able to respond interactively to incorrect or undesirable user actions. It can also make the program state available for inspection.

The integration of the tools allows IPE to present the programmer with a uniform view of the program in terms of its source form. The program is manipulated through a syntax-directed editor, and its execution is controlled by a language oriented debugger. The debugging actions are embedded in the supported language and are invoked by editing the program. Other tools are applied automatically by IPE at the appropriate times. These tools and their representation are not visible to the user, the only (thus uniform) interface is the user interface of the editor. During the construction and manipulation of the program, the programmer focuses on a small piece of the program at a time. The program modifications cause the system to incrementally compile those pieces and incorporate them into the executable version of the program.

One of the goals of IPE is to explore whether an IPE system can be supported based on compilation technology only (i.e., no interpretation), but still providing facilities traditionally found in interpretive systems. By compilation technology we refer to the translation of the source program into a lower level representation such as machine code, which is executable separately from the source representation. In many compiler systems to date the debugger provides an interpreter for a limited subset of the language, in contrast with IPE which implements the debugging facilities based on compilation. In other words, we make the flexibility as it is found in interpretive systems available to the programmer who uses a compiler-based system.

Interactive programming environments that support compiled code pieces have been implemented before. Interlisp [1] is a very sophisticated programming system for Lisp based on interpretation. It allows program pieces to be compiled, but they are linked into the execution by the interpreter. Similarly debugging is done through interpretation. Other systems such as the Cornell Program Synthesizer [2] are also based on interpretation, but support compiled expressions.

Because IPE provides support not only for experimental programming but also long-term program maintenance, a strongly

0098-5589/81/0900-0472$00.75 © 1981 IEEE

typed Algol-like language with facilities for abstraction and modularization was chosen. A perfect choice would have been Ada, but as the language is still going through the final modifications and compilers do not yet exist for it, we have chosen instead GC, a type-checked variation of C with module structure that runs under UNIX™ at Carnegie-Mellon University [3].

IPE as reported here provides a comfortable environment for a single programmer working on a single program. However, its design has been influenced by the fact that it is a building block of a more general software development environment. Such an environment is being designed and built at Carnegie-Mellon University in the *Gandalf* project [4], and IPE is part of that project [5]. Gandalf, in addition to IPE's facilities, provides support for managing a project that involves the interaction of several programmers [6], and for the manipulation of system compositions and version control [7].

The rest of the paper is organized as follows. In Section II we describe traditional methods of programming in compiler systems and relate to them the approach taken in IPE. Section III contains a discussion of various design and implementation issues dealt with in IPE. Finally, in Section IV we draw some conclusions and describe the current state of the IPE implementation.

II. Program Modification Cycle

As a programmer goes from specification of programs to their implementation, many tools are normally used. The important issue here, however, is not the tools that are used, but the problems that the tools address. For example, a typical editor does not manipulate programs, but only manipulates text.

There are four problems that programming environment tools must help the programmer solve. First, there must be a method for the programmer to enter and modify programs. Second, there must be a way of ensuring the syntactic and semantic correctness of programs. Third, an executable form of the program must be created. Fourth, there ought to be a process that can help the programmer debug programs.

In the traditional methods these tools are largely independent of each other, have different user interfaces, and use different representations for the programs. In the following sections we contrast the traditional methods with the IPE approach in which the programmer is presented with an integrated and interactive system.

A. Traditional Methods

Each of the programming environment problems mentioned above has been addressed many times since people started to program computers. Almost all the solutions, however, fit into a standard tools cycle. The variations on the tools used and the order of their use are slight, so the description that follows is representative of most traditional compiler based programming environments.

The traditional tools are the editor, the translator, the linker and loader, and the debugger. The order in which these tasks are usually applied is: enter/modify, compile, link and load, debug. This cycle is repeated until the program appears to be correct. The remainder of the section describes these tools in more detail.

1) The Editor: The editor is used in a programming environment as a way to initially enter and subsequently modify programs that are being developed in a particular programming language. The drawback with traditional editors is that they are general purpose text manipulators. The editor usually has no knowledge of whether the text being entered represents a program, a document, or a poem. If it has some idea, it is generally very limited. This ignorance inherently causes the need for parsers that determine if the text which represents a program is syntactically correct. The parser is usually part of the translator, which makes the checking process very costly.

2) The Translator: The basic job of the translator is to transform program text into another representation of the program. In a compiler this representation is usually machine or assembly language, while in an interpreter it is often an intermediate language.

In traditional systems there are two tasks required for this transformation. First, the textual representation of a program must be analyzed for syntactic and semantic accuracy, a process that is necessary because of the inability of the editor to support a programming language. The resulting parse tree is then checked for consistency of the language semantics, which includes type checking, parameter checking, and operand coercion.

The second task is the actual translation of the program into a machine representation, such as assembly, object, or even microcode. This is done by a traversal of the parse tree, and is often combined with some semantic checking. In the process of generating code, optimizations are often applied at various points in order to improve the quality of the produced code. However, during program development, nonoptimizing code generators are often used.

3) The Linker and Loader: Programs, even small programs, are often broken into several smaller pieces. It is usually convenient to keep these pieces in separate files and work on them independently. The amount of program text to be checked is kept small. However, there are often links among the pieces that must be maintained. These are known as external references and information about them is generated by the translator.

In a traditional system it is the job of the linker to resolve these external references so that previously translated program pieces can be combined to form a complete executable form of the program. Because no assumptions are made about other program pieces already being translated, linking is a separate pass in the tools cycle, requiring all translated program pieces to be processed for reference resolution. The loader's job is simple once the linker has resolved the external references. All it has to do is actually place the program in memory so that it can be executed correctly. Traditionally, whenever any kind of change is made to a loaded program (i.e., a piece is altered and recompiled), linking and loading has to be done again. Some systems such as Multics [8] reduce the effort of the linker and loader by providing a dynamic linking mechanism as part of the operating system.

4) The Debugger: The debugger is used to help a programmer observe the execution of a program. The purpose of this

is to detect erroneous program behavior and to locate its cause. A very simple form of debugging support is symbolic dumps. A more advanced form is the interactive debugger that allows programmers to dynamically set trace and break points at arbitrary locations, to inspect and alter values of variables and data structures, and possibly even to use conditional debug functions (e.g., break whenever $x = 0$). While debuggers allowing these and possibly other actions are often available, most of them do not work at the source code level. Variable and procedure names can be referred to symbolically, but for the rest the programmer must deal with the machine representation.

Source level debuggers are mostly found in conjunction with interpreters. Where compiler systems provide source level debugging, it is accomplished with a simple text line to machine representation mapping. The unit for break and trace points is a text line rather than a statement or expression. The compiler in such a compiler system in general does not optimize code because certain optimizations (e.g., code motion) make the mapping between the two representations difficult to maintain.

B. The IPE Approach

The approach to program development supported by IPE has some important advantages for both the programmer and the IPE implementor. The integration of tools and their automatic application at appropriate times provides a single and uniform user interface. The programmer deals with his program uniformly in terms of language constructs. Modularity of the language restricts the accessibility at any point in the program. The programmer has controlled access to a subset of the program's procedures and objects, and thus can cause less damage to unrelated program pieces.

Modularity also restricts the scope for semantic checking, making it much more simple and efficient. The fact that a tool resides in a well-defined environment, i.e., specific tools exist and they are applied at appropriate times, can be used to simplify the design and implementation of that tool. To enhance this cooperation all tools share a common program representation—the syntax tree representation. This avoids the usual necessity of constructing one representation in terms of another (e.g., writing a syntactically correct program using a line editor).

The tools cycle in IPE, called *incremental program modification mechanism*, is very different from the traditional tools cycle. The programmer has a uniform view of his program in terms of source code. The program is manipulated through a syntax-directed editor and its execution is controlled by a language-oriented debugger. Other tools, such as the translator and the linker/loader, are not visible to the user. They are automatically invoked to keep the tree representation and the (invisible) machine representation consistent.

1) Syntax Directed Program Editor: Modifications to the program can be made through the editor during construction of the program, and any time the program execution is suspended. Language constructs (such as variables, operators, expressions, different types of statements) can be added, modified, or removed. The programmer communicates with the editor in terms of language constructs.

The editor constructs and manipulates a program tree directly. However, the programmer appears to construct the program by inserting templates that represent different language constructs and then filling the "holes" of those templates with other templates. Since the editor knows which constructs are valid at any given point, it allows the programmer to insert a language construct only in a place where it is syntactically correct.

For example, instead of typing the character sequence for an **if** statement, the programmer calls on the template "**if.**" The result is the insertion of

```
if (<expression>)
    <statement>
else
    <statement>
```

at the current program position, provided that this construct is syntactically correct in that context. The current program position is advanced to <**expression**> so that it can be similarly expanded. Note that the editor provides all the necessary keywords, separators, terminators, and all the other "syntactic sugar" required by the language like the parentheses around the <**expression**> construct that are required by the C language syntax. Problems such as misspelled or nonmatching keywords cannot occur because the language constructs are inserted by the editor and not by the typist. The editor relieves the programmer from the worries of the syntax constraints imposed by the language.

The editor internally represents the program as a *syntax tree*. Each template corresponds to a node of a certain type in the tree. The holes of the template are the offsprings of the node. They will be filled in as subtrees representing the expansion of those holes. Fig. 1 shows the representation of the "**if**" statement of the example above. To present the programmer the text of his program the editor uses an *unparser* that translates the syntax tree into human readable text. As part of its task this unparser does the pretty-printing of the program. Thus, the programmer actually constructs and manipulates a program tree without necessarily being aware of it.

The programmer interacts with the editor through language commands and editing commands. Language commands are used to construct new templates (e.g., the call for an "**if**" template). Editing commands are used for manipulating the program tree (e.g., delete a subtree). Even though most of the debugging interaction is expressed in terms of language and editing commands on the program tree, the editor command list is extended to include two commands that control execution, i.e., start or continue execution and interrupt execution.

The editor invokes the semantic checking routines while the programmer is constructing or modifying his program and informs the programmer of any semantic inconsistencies. The semantic checking goes on while the programmer is "thinking" at the terminal. When the programmer finishes the editing session, the program is syntactically correct and semantically checked, but it could be incomplete, that is, some "holes" may be left unexpanded. The editor does not enforce semantic correctness. It informs the programmer of errors and allows

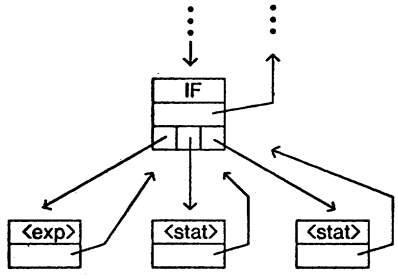


Fig. 1. Representation of an IF subtree.

their correction. Code will not be produced for a program that is not semantically correct. Semantic correctness in this context means correctness with respect to the programming language semantics.

The idea of syntax-directed editing is not new. Some previous and related efforts include the following.

- The Emily System [9] was one of the earliest efforts with syntax-directed editing. The programmer constructed a program by selecting a BNF production to replace the current nonterminal. Emily was very slow and editing was very difficult. In order to compile a program, the programmer had to produce a textual representation of the program which was parsed by the compiler. The internal structure of the program was not taken advantage of.
- Mentor [10] is a structure editor oriented towards Pascal. Mentor incrementally parses the program text to create syntax trees. It allows structured modification of programs. The syntax trees are not used by the compiler. It must parse the programs from their textual representation.
- The Cornell Program Synthesizer [2] is a structure editor that supports construction and manipulation of syntax trees. At the expression level, however, the programmer inputs his program as plain text which is then parsed. The synthesizer is implemented for PL/CS a small subset of PL/I in a microcomputer, i.e., it is geared towards the construction and execution of small programs.

By comparison, IPE performs structured editing at all levels including expressions. The current position in the program tree is clearly marked by highlighting the program text that represents the subtree. The abstract syntax description of a language defines names for operators that are specific (and mnemonic) for that language. Editors can be automatically generated from these grammatical descriptions. Finally, the compiler uses the internal tree representation to generate code, i.e., avoids parsing and syntax analysis.

2) The Common Representation: The syntax tree constructed by the editor has been chosen as the common representation of programs. There are two types of nodes in the syntax tree: terminal nodes (or leaves) and nonterminal nodes.

Terminal nodes are used to represent variables, constants, some static language elements (e.g., data type names), and unexpanded program constructs (placeholders or "holes") which will be substituted by the correct subtree in the process of constructing a program. The node for variables contains information about the symbol table. It also provides some space to put semantic information like the type of the variable and references to other occurrences of the same variable.

Nonterminal nodes describe subtrees of the program corresponding to control flow constructs and data definitions in the language (e.g., an **if** construct). The information available at the node includes the type of language construct, references to the parent node, and to its offsprings. There are two classes of nonterminal nodes, one with a fixed number of offsprings, and the other with a variable number of offsprings (represented as a linked list for constructs like a compound statement). In addition, space for semantic information and mapping information from the code generator is provided.

Fig. 1 illustrates the representation of the **if** statement example mentioned in Section II-B1). The three offsprings are terminal nodes representing unexpanded nodes.

In the next sections we describe the remaining mechanisms and tools that are supported by IPE. They all interact with the common program representation. They are grouped into program translation, i.e., the update of the static program representation, and debugging, or analysis of dynamic program behavior.

3) Incremental Program Translation: During the process of program construction and modification the semantic correctness is checked. This checking is invoked automatically on an incremental basis whenever the programmer completes changes to a program piece (e.g., leaves the scope of a procedure with the display cursor). Semantic correctness is not enforced, that is, a programmer can use a variable before declaring it. However, code is only generated for semantically correct program pieces.

Similar to semantic checking, code generation and link/loading are invoked automatically. This activity is invisible to the programmer, with the exception of possibly noticing it in the system response. In order to keep the response time at a reasonable scale, IPE processes the program incrementally. Whenever the programmer completes modification of a program part, e.g., a procedure, code is generated from its abstract syntax tree. In the process all references can be resolved for the generated code. The executable code for the modified procedure is then deposited in the execution image by a loading mechanism that replaces program parts in the execution image statically, i.e., before execution starts or resumes. The replacement does not affect other program parts in the execution image [see Section III-B1)].

The incremental program modification mechanism supports execution of incomplete programs. All semantically correct procedures have executable code loaded in the execution image. For procedures with semantic errors IPE loads a code stub. Execution of the program can be started at any time. If the control flow encounters a code stub execution is suspended. At this point the programmer can correct the procedure, causing the code stub to be replaced by the actual code for the procedure, and resume execution. This leads us to the debugging support provided by IPE.

4) Language-Oriented Debugging: The debugging facility of IPE is implemented without interpretation. This is achieved by integrating the debugger with the incremental program modification mechanism. By doing so the debugging actions can be embedded in the program through extensions of the programming language. For example, a conditional breakpoint

is set in a procedure through the syntax-directed editor by inserting the appropriate construct in the abstract syntax tree of that procedure. As a result of the insertion the procedure is automatically processed by the incremental program modification mechanism, and the debug statement is reflected in the execution image. The required language extensions, however, are limited. Some languages already contain constructs that can act as debug statements, such as *assert* in Euclid or Ada and *pause* in Fortran.

By integrating the debugger with the incremental program modification mechanism, the programmer is presented with a uniform interface to IPE. All interaction with IPE is performed through the syntax-directed editor in a language-oriented manner. The implementation of the debugger around the abstract syntax tree allows the debug facilities to take advantage of the language knowledge that is embedded in the tree. The content of variables is displayed according to their data type. The location of execution is indicated by positioning the program cursor in the tree, which is useful for execution tracing. The programmer can evaluate any legal program part, such as an expression, by defining it with the syntax-directed editor. If it is semantically correct code is generated and it is evaluated by executing the temporarily loaded piece of code.

The debugger implementation based on the abstract syntax tree also lends itself to the provision of more sophisticated debugging actions. One example is the runtime checking of assertions. This idea has existed for some time (e.g., in Euclid [11]). Assertions are provided by the programmer as part of the program. They are in general limited to expressions that may include calls to functions without side-effects. We attempt to expand the expressive power of assertions beyond that of many programming languages. In order to express certain verification conditions, constructs such as quantifiers and previous values are introduced. Initially, no code is generated for assertions, but at any time during program execution the programmer can request them to be enabled. Assertion code is then added in the machine representation, and the validity of the assertions is checked. This mechanism does not guarantee correct programs, but is a small step in the direction of producing verified programs.

Other debugging actions make use of the data flow and control flow information in the program tree. An example of a debugging action in this class is the tracing of a *crazy* variable, i.e., locating the place where a variable is assigned an erroneous value. This is done by attaching a condition (e.g., an assertion) to a variable and having it automatically checked whenever the variable is changed.

In summary, a language-oriented debugging facility with quite sophisticated functions can be provided without requiring a separate complex system such as an interpreter, by basing the implementation on the abstract syntax tree. The structure editor serves as a uniform language-oriented user interface. The incremental program modification mechanism allows debugging actions to be specified and enabled dynamically. Furthermore, debugging actions do not impose any overhead on the execution when they are not enabled.

III. Design and Implementation Issues

In this section we discuss some technical issues that have arisen in the process of designing and implementing IPE.

A. The Editor

As mentioned earlier, the editor knows how to build programs in a specific programming language. The major advantages of this constructive approach are that programs are edited in terms of the programming language constructs, and that users cannot enter syntactically incorrect programs. Hence, the first few compiles just "to get the semicolons right" are no longer necessary. The whole process of translation should be much more efficient than the normal method of lexical and syntactic analysis, since the translation of character strings to lexemes and from sequences of lexemes to syntactic units are no longer necessary. Parsers are no longer needed.

The programmer can manipulate the program tree through language commands and editing commands. Language commands facilitate the building of program pieces in terms of language constructs by expanding the "holes" in the program templates. The "holes" in the templates are called *meta-nodes*, that represent nodes that have not been expanded. Meta-nodes have a name (also helpful for display purposes). By default they get assigned the name of the set of language constructs (also referred to as language operators) that can be applied at that point (e.g., the set of statements). The programmer may rename such meta-nodes in order to remember the name and return to them at a later time, when he wants to expand them. After every language command the current tree position is advanced to the next meta-node. Editing commands provide a facility for more general tree manipulation.

As part of the language specification several *unparsing schemes* are given for every language construct. This provides an easy way to implement several pretty-print formats for the same programming language. It also allows to present different "views" of the same program by limiting access to parts of the abstract syntax tree (e.g., one view that has the specification and implementation parts of a module and another that only has the implementation part).

Figs. 2 and 3 illustrate a sample editor session for the process of creating a "**for**" loop statement in C. The first column gives the commands typed by the user, the second column shows how the terminal display would look like after the command is executed, and in the third column the syntax tree structure is presented.

Note that commands do not need to be typed fully, the programmer only needs to type those characters needed to unambiguously determine the command. Some commands have synonyms for convenience (e.g., PLUS has "+" as a synonym). Variables are preceded by a single quote to differentiate them from commands.

Some of the editing commands available in the editor are as follows.

Delete causes the deletion of the subtree that is at the current program position. A meta-node is then inserted on its place.

Clip removes a subtree from the program tree, but does not

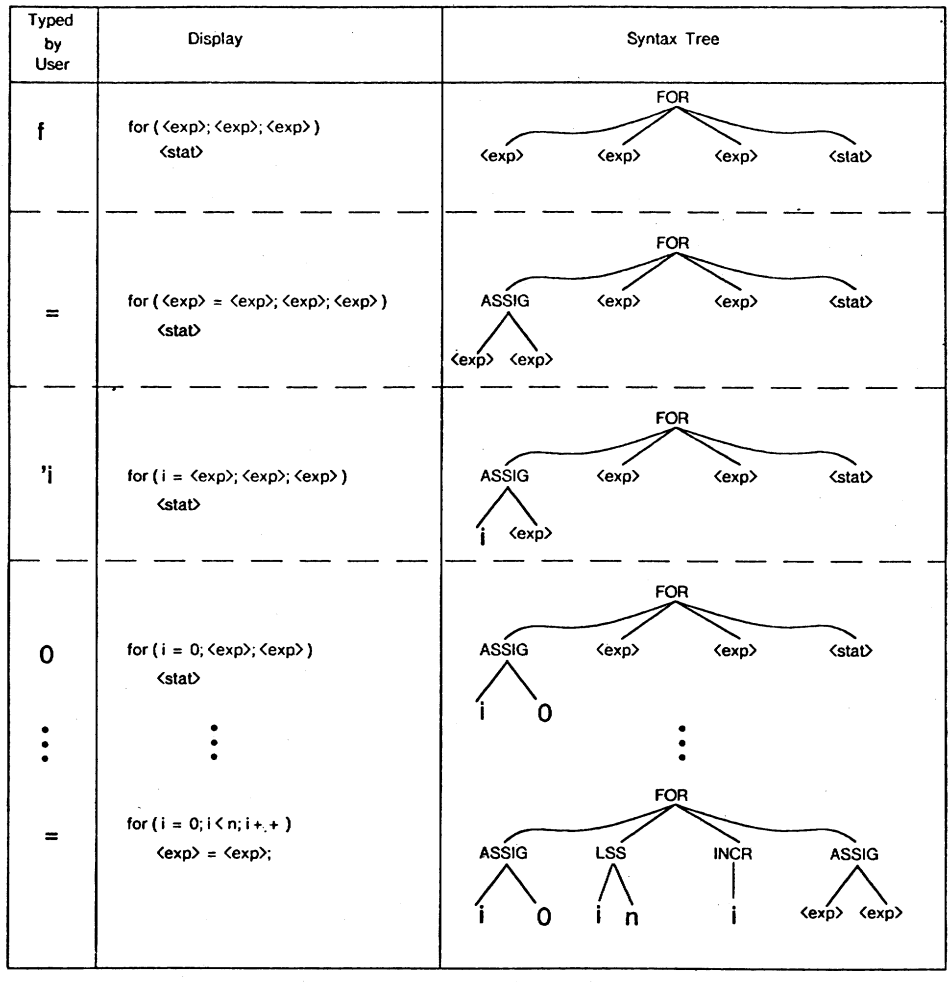


Fig. 2. A sample editor session.

throw it away. This command prompts for a name under which the subtree can be referred to for future use. A meta-node is put in its place.

Insert prompts for the name of a previously clipped subtree and inserts it in the place of a meta-node as long as the root of the subtree is of the correct type (e.g., the insertion is syntactically correct).

Write saves the program tree by writing it into a file, so that it can be retrieved later. This command allows the programmer to save his program for later development.

Read retrieves a program tree from a file and places the current tree position at the first meta-node or at the root if there are no meta-nodes.

Find moves the cursor to a node that matches a pattern.

Nest and *Transform* provide more sophisticated program manipulation, always guaranteeing the syntactic correctness of the program.

There are also commands to "move" around the tree: they move the current tree position (e.g., to the "father," to the "first son," to the "siblings," or to some subtree specified by a pattern, etc.). The current tree position is always highlighted with an "area" cursor on the terminal display.

In addition, there are other commands that provide on-line help facilities to inform the programmer about the available commands (e.g., which are the valid language commands at the current tree position?). In a sophisticated display device this information could be presented in a menu-like format.

As mentioned above, syntax-directed editing has a major impact on the implementation of programming environments because it makes lexical analyzers and parsers obsolete. In addition, it may also influence reference manual writing and even language design. The manual would no longer have to describe if a semicolon is a statement separator or a terminator and the language designer would not necessarily need to decide this. Emphasis will then be placed on the language constructs and the facilities provided by the language.

Some language ambiguities (e.g., the "dangling else" problem of some languages) disappear because with the constructive approach it is clear what the programmer means (e.g., to which **if** statement the **else** part belongs), and there is no need for a parser to try to figure it out.

Another traditional problem of programming languages has been the correct parenthesization of expressions given the operator precedences of the language. In the syntax-directed editor the programmer constructs the expression tree, i.e., specifies the order of operator application directly. As a part of the unparsing mechanism, the editor generates the appropriate parenthesization for the text form that is displayed on the terminal screen given the precedence of operators.

While we believe in the merits of syntax-directed editing,

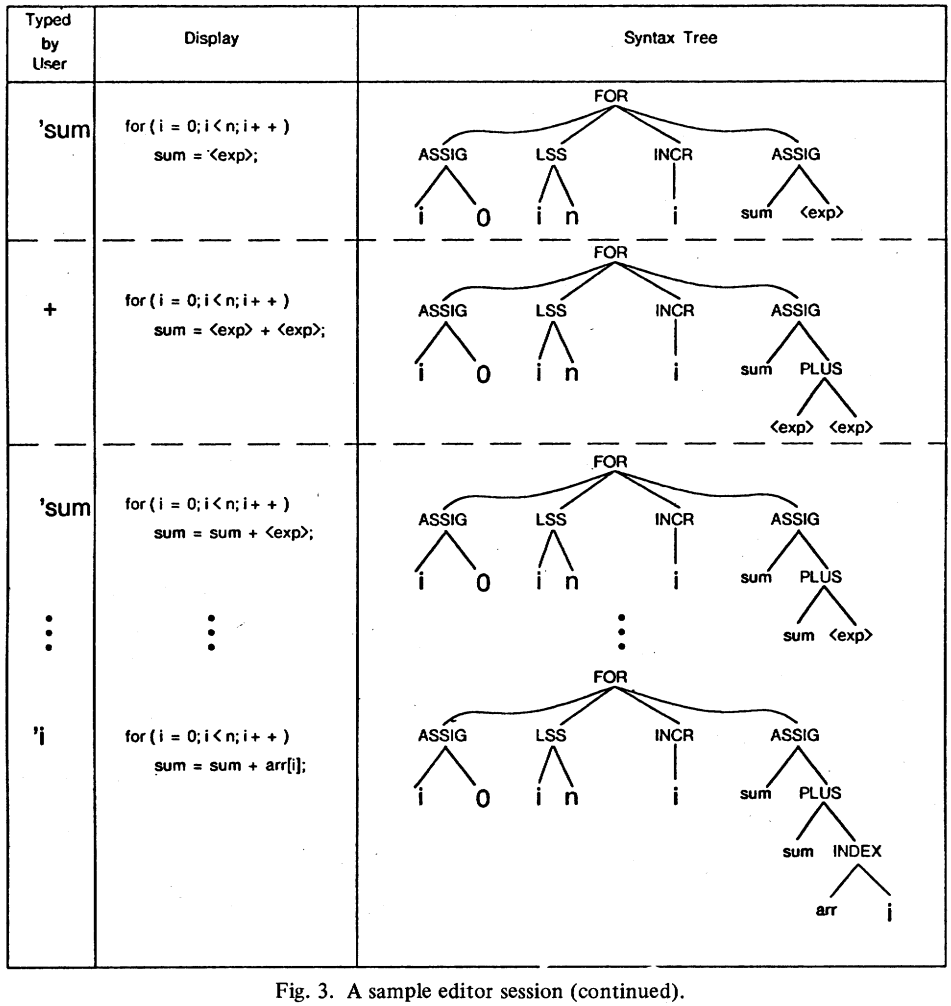


Fig. 3. A sample editor session (continued).

there are, nevertheless, some possible disadvantages. New users of the editor will find some difficulties in getting used to the structured editing approach. They would have to go through a learning period before feeling "at ease" with the editor. Expression entry and editing is an instance in which structured editing may be more difficult than text editing. Similarly, program modification may be perceived to be awkward, but sophisticated editing operations such as nesting and transformations overcome the problem. Another disadvantage of the syntax-directed editing approach is that programs are stored in tree form. In order to use the editor with existing programs a parser must be provided to convert the program text into tree form.

1) The Editor Viewed Independently from IPE: The editor can also be viewed separately from IPE. A *syntax-directed editor generator* has been developed. It permits the automatic generation of a constructive editor for a formally described programming language. The syntactic formalism describes languages in terms of an abstract syntax notation. It defines the language constructs and the sets of valid language constructs and the sets of valid language constructs for every "hole" in the templates. In addition, formatting (unparsing) information for the display of a construct, the precedence of the construct, a synonym with which the construct can be invoked as a language command, and the name of a routine that is invoked by the editor upon an editor operation on the given node, are specified for every construct in the grammar. Multiple unparsing schemes may be specified, which permits the definition of different textual program "views" from the same program tree. These views may show the program at different levels of abstraction, with different "syntactic sugar" and formatting. The language description clearly separates between the abstract syntax that defines the internal structure of the program, and the concrete program representations that are displayed on the screen.

A preprocessor is applied to such a description and it produces the internal tables that are used by the editor as its "knowledge" of the language. The language designer can also define extended editor commands that will be tailored to the specific language or system (i.e., in IPE the extended commands are continue and interrupt). This process has been applied successfully to several very different types of languages: to a subset of Ada, to GC on which IPE is currently based, to *Alfa* a non-Algol-like applicative language designed by Habermann [12], to the system composition and version maintenance language of Gandalf [7], and to the gra..mmatical description itself.

B. The Program Representation

The IPE system internally maintains two representations of a

program, the tree representation and the *machine* or *executable* representation which is interpreted by the hardware during program execution. The first representation is the syntax tree of the program. One choice for the representation of the syntax tree was TCOL, especially $TCOL_{Ada}$ [13]. It is a symbolic tree representation in text form that has been accepted as one standard in the Ada language development. We decided against using that representation directly because of the conversion cost into an internal representation. We have chosen an internal representation that comes very close to an internal representation of $TCOL_{Ada}$, and we are able to generate a TCOL representation if necessary.

The second representation is generated automatically from the first through a translation step. This separation of a program into two representations allows a *host/target* approach, i.e., the execution of the program may be performed on a machine different from the machine running the IPE system, as required by Stoneman [14]. Since all IPE tools primarily work on the tree representation, very little IPE support must be provided on the target machine. In order to incorporate a new target machine into IPE, the target machine runtime support and a code generator for the target machine on the host must be constructed. Therefore, the dependency on the target machine is limited.

The changes made by the programmer in the tree representation of the program must be reflected in the machine representation. The behavior of the executing program must be consistent with what the programmer expressed in terms of the programming language. Since changes are made incrementally to the program tree, they also are reflected incrementally in the executable representation. However, the cost of this incremental program modification must be kept small; otherwise it will be noticeable to the programmer in the system response.

1) The Machine Representation: The structure of the machine representation has a strong influence on the cost of an incremental program modification. On one hand, a certain complexity in the supported structure, e.g., segmentation, permits a simple replacement mechanism. On the other hand, the representation should be supported directly on existing conventional machines without a software simulator.

The procedure has been chosen as the replacement unit of the program. Procedures have fixed entry points that refer to the actual location of the code. They are assigned when the procedures are specified. All procedure calls are made indirectly through these entry points. Global data are placed in an area that is separate from (and unaffected by) the replacement of program pieces.

Code generation is invoked on a semantically correct subtree representing the procedure to be compiled. Since the procedure is semantically correct, all called procedures must have been specified, i.e., entry points have been assigned to them, and all referenced global objects must have been declared, and therefore, space has been allocated for them. Therefore, all global references are known at code generation time. All local references within the procedure can be produced with all external references resolved. A separate linking step is not necessary.

When the machine representation of a procedure is replaced, the new copy may not fit into the space of the old one. Therefore, it must be placed in some free memory space. Since memory space is often limited, released memory space must be recovered and the allocated space compacted. This may require moving the machine code for some of the procedures. In order to make this moving possible without requiring a relink of the moved program pieces, restrictions are imposed on the use of different addressing modes of the machine. References are kept invariant from the physical location of the code, if all references within the moved code piece, i.e., local references, are relative to the code piece (e.g., PC relative), and all references outside the moved code piece, i.e., global references, are absolute. Thus, code can be compacted through a simple block-move operation. This garbage collection problem may not be relevant if an underlying machine with a more complex memory structure than linear memory space is available (e.g., a segmented machine).

A comparison with conventional compilation techniques, including separate compilation, shows that the cost of generating code for a modified program part is greatly reduced, thus keeping the response time small. Since the program is already in tree form, text does not have to be processed: lexical and syntax analysis is not done. Semantic checking is performed incrementally while the programmer modifies the tree through the editor. Code is generated and loaded incrementally. The link-editing step is merged into the code generation phase. Thus, the processing cost consists of the cost of code generation and loading of the modified program part.

2) The Mapping: For debugging purposes, a mapping between the tree representation and the machine representation is required. This mapping allows the flow of execution in the machine representation to be traced in the tree representation. The mapping, however, does not need to be complete. Only certain tree nodes must have an exact mapping in order to determine the exact execution state in both representations (e.g., routine entry, return addresses, debugging stoppage points). For other program parts only an approximate mapping is needed. For example, signals from program execution that are not handled by a debugging action, are only indicated to have happened in a certain program region.

The mapping is produced by the code generator. It may be either locators for tree nodes in the machine representation or references to code pieces corresponding to tree nodes stored in the tree representation. Optimizations are applied by the code generator in order to improve the quality of the machine representation. However, they may affect the mapping of the program. Optimizations that transform the program tree cannot be employed at mapping points, and optimizations that cache program state in machine registers must update the cached values in memory at those points. Because the mapping is incomplete and the mapping points are sparse, optimized code is produced for large parts of the program, and debugging is still supported.

C. Program Execution

So far we have discussed mechanisms in IPE that maintain the static part of a program. This assumes that program execu-

tion is restarted after a modification. This is, however, an unacceptable approach for IPE, especially because debug actions are implemented through the incremental program modification mechanism. In order for the programmer to be able to dynamically debug programs, continuation of execution must be supported at least for certain program modifications.

1) Continuation of Execution: IPE supports three forms of continuation of execution. The first is resumption of execution, i.e., continuation at the point where execution was suspended. The second form of continuation is unwinding of the stack of procedure activations. In this case the state of variables is not changed. The third form of execution continuation involves restoration of the execution state to a previous point.

When the programmer modifies a suspended program IPE attempts to preserve the execution state so that execution can be resumed. Resumption of execution can be permitted if the two program representations and the execution state of the program are in a consistent state. Consistent state means that the program shows the same behavior on resumption of execution after a change as it would when restarting the execution with the new executable version.

Program changes to the control flow are easily dealt with if they do not involve active procedures. The modified procedures are replaced by the incremental program modification mechanism and have no side-effects on the execution state. However, when replacing the executable representation of an active procedure, the placement of the code body may invalidate the return address on the activation stack and require an update. Similarly, garbage collection in the executable representation affects the return address of all active procedures that are being moved. Since return addresses are located on the activation stack, they are retrievable and can be updated.

Certain program modifications do not affect the execution state other than moving the activation points of active procedures, i.e., return addresses and current program counter. These are modifications to a part of an active procedure whose subtree does not contain an activation point. An example is the addition of a statement. All debug statements fall into this category.

For other modifications the execution state cannot be adjusted to the change. This is the case when the change to the program affects the dynamic control flow. For example, procedure A calls procedure B. Both are on the activation stack and execution is suspended in B. The change is to remove the call to B in A. What is the execution state from which execution can be safely resumed? Thus, resumption from within B does not make sense.

One alternative that IPE offers to the programmer is to unwind the nesting of active procedures until the modified active procedure is removed from the stack. The programmer can then continue execution which will repeat the last procedure call that was removed from the activation stack. Execution unwinding does not restore the state of variables to their previous values. Therefore, the programmer must decide whether continuation is safe after unwinding or whether certain values (e.g., semaphores) must be restored explicitly.

A second alternative, other than starting execution from scratch, is to restart execution at a safe point (e.g., the call to the procedure with the change). This requires resetting the current execution state to the execution state at the time of the call. Depending on the definition of the execution state, this may be a hard problem. It is related to work done on recovery mechanisms (e.g., recovery blocks [15], stable variables [16]). In IPE we take a conservative approach. Initially, there is no automatic facility to reset program execution other than starting it over. However, the programmer may explicitly specify and enable checkpoints to which the execution state can be restored. The cost for checkpoints is high, and the programmer must anticipate the program location to which restoration is desired.

2) Debugging: Debugging actions in IPE are implemented through the incremental program modification mechanism. This approach has several advantages. The debugger works on the tree representation, i.e., in terms of the source language. Debug statements are actually recorded in the tree and are visible to the user. Their implementation in the machine representation is done by the code generator. Therefore, the debugger has very limited knowledge of the machine representation. The evaluation of expressions in context is implemented in a similar fashion by temporarily adding a subtree for the expression and generating code for it. Potentially, the display of variable contents can be implemented by using the expression evaluation mechanism. This implementation, however, is slow. Therefore, a special variable display mechanism is implemented to provide access to the program state in a more efficient manner.

The approach taken to implement the debugging actions also has some disadvantages. First of all the responsiveness of the debugger is dependent on the response time of the incremental program modification mechanism. Therefore, that mechanism is the critical path of the whole IPE system. Some debugging functions are more difficult to implement. One such function is single stepping, a mechanism found in many traditional systems. It requires a substantial amount of retranslation overhead if it is applied to large parts of the program. As an alternative, however, IPE provides more sophisticated debugging actions that are more tailored towards the actual debugging goal.

IV. Conclusions

This paper presents IPE, an incremental programming environment, being designed and implemented by the authors at Carnegie-Mellon University. This system gives programming support to a single programmer working on a single program. Even though it is presented as a system in itself, IPE is part of Gandalf, a more general software development environment project. For Gandalf, IPE is being extended to incorporate support for description of larger systems, multiple versions of programs, and management of several programmers on the same system.

We consider the major impacts of IPE to be its integrated approach and the fact that it provides an interactive environment based on compilation technology. The programming environment is now viewed as a single system rather than as a set of independent tools. There is a smooth transition be-

tween the tools, and some tools are not directly visible to the programmer because they are automatically applied.

At all levels of the program development the programmer deals with language constructs. The programmer does not have to worry about the syntax constraints of the programming language any more, nor has he to be aware of the different components of the environment. By making it easier for the programmer to construct and debug programs, it is very likely that the quality of his programs will be improved and the programming time will be reduced.

IPE is based solely on compilation technology. No software interpreter is provided. Compilation has the advantage that a high-level program representation is transformed into a separate program representation (machine code) for a computer which is not necessarily the same as the host machine. Therefore, IPE may be able to support several target machines with most of the IPE system residing on one host machine.

A syntax-directed constructive editor allows the programmer to build and change programs in terms of language constructs. This reduces the amount of typing and eliminates programming errors such as missing semicolons. The editor uses its knowledge about the programming language and permits only syntactically correct programs to be constructed. Despite the fact that the editor has knowledge of the language, it is language independent in the sense that the editor is automatically generated from an editor kernel and a formal description of the language. The program is internally represented by an abstract syntax tree, from which the text form is dynamically generated for display to the programmer. This syntax tree is the common program representation for all tools in IPE. Semantic checking, translation, and debugging work on the syntax tree.

The translation phase uses the fact that code is produced only for semantically correct program pieces to merge the linking step into the code generation. As the programmer is incrementally changing the tree representation of the program, the IPE system incrementally updates and maintains an executable version by automatically applying the translation phase to program pieces and incorporating them on the target machine. The debugging facility of IPE also works on the abstract syntax tree. It is implemented using the incremental program modification mechanism, i.e., incremental update, translate and load, and, therefore, does not perform software interpretation. The code generator provides the mapping from the tree representation to the machine representation. Some optimization can be applied during code generation because only certain points in the tree require an exact mapping, even for debugging. Since the abstract syntax tree is the primary program representation for the debugger, more sophisticated debugging actions in terms of the underlying programming language can be provided.

An IPE prototype is running under the UNIX™ operating system on a DEC/VAX™. At the time of this writing the state of implementation is the following. The editor generator has been implemented and an editor that supports GC has been generated. Semantic checking is performed by the incremental code generator for GC. An incremental loader and a basic debugging facility have been implemented. The debugger is currently being extended to include sophisticated functions. To implement IPE we have used an incremental system that already incorporates some of the program development philosophy supported by IPE [17] (i.e., it uses the syntax-directed editor for GC).

The IPE prototype implementation indicates that an interactive, incremental programming environment based on compilation is feasible. The editor generator has shown to be a powerful tool for experimentation with structured editing of different languages. The ability to specify multiple unparse schemes for one abstract syntax tree permits the definition of program views of different abstraction levels, e.g., list of modules, module specification, module specification and implementation, and pretty-printing tailored to the individual programmer.

Practical experience in the use of IPE for the construction of larger programs will be gained over the next months, as it will be used as the tool to continue the development of IPE in the context of Gandalf.

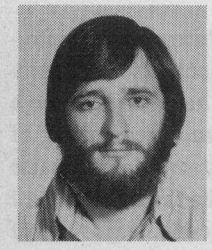
Acknowledgment

The authors would like to thank the members of the Gandalf project at Carnegie-Mellon University for their helpful discussions. They would also like to thank the referees for their constructive comments which improved the presentation of this paper.

References

[1] W. Teitelman, *Interlisp Reference Manual*. Xerox Palo Alto Res. Cen., 1978.
[2] T. Teitelbaum and T. Reps, "The Cornell program synthesizer: A syntax-directed programming environment," Dep. Comput. Sci., Cornell Univ., Ithaca, NY, Tech. Rep. TR80-421, May 1980.
[3] P. H. Feiler and R. Medina-Mora, "The GC language," Gandalf Internal Documentation, 1979.
[4] D. S. Notkin and A. N. Habermann, "Software development environment issues as related to Ada," Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep., 1979.
[5] A. N. Habermann, "An overview of the Gandalf project," *CMU Comput. Sci. Res. Rev. 1978-79*, 1980.
[6] —, "A software development control system," Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep., 1979.
[7] A. N. Habermann and D. E. Perry, "Well-formed system compositions," Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-CS-80-117, Mar. 1980.
[8] E. I. Organik, *The Multics System; An Examination of its Structure*. Cambridge, MA: M.I.T. Press, 1972.
[9] W. J. Hansen, "Creation of hierarchic text with a computer display," Ph.D. dissertation, Dep. Comput. Sci., Stanford Univ., Stanford, CA, June 1971.
[10] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang, "Programming environments based on structured editors: The MENTOR experience," presented at Workshop Programming Environments, Ridgefield, CT, June 1980.
[11] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. L. Popek, "Report on the programming language Euclid," *SIGPLAN Notices*, vol. 12, Jan. 1977.
[12] A. N. Habermann, "Notes on programatics and its language Alfa," private communication.
[13] B. Schatz, B. Leverett, J. Newcomer, A. Reiner, and W. Wulf, "$TCOL_{Ada}$: An intermediate representation for the DoD standard programming language," Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep., 1979.
[14] Dep. of Defense, "Requirements for Ada programming support environments, 'Stoneman,'" Feb. 1980.
[15] B. Randell, "System structure for fault tolerance," *SIGPLAN Notices*, vol. 10, June 1975.

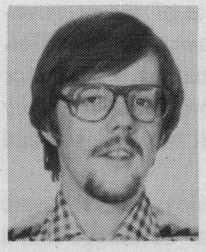
[16] B. Liskov, "Primitives for distributed computing," Distinguished Lecture Series, Carnegie-Mellon Univ., Pittsburgh, PA, 1980.
[17] P. H. Feiler, "IPC system version 1," Gandalf Internal Documentation, 1979.



Raul Medina-Mora (S'81) was born in Mexico City, Mexico in 1953. He received the B.S. degree in applied mathematics (Actuario) from the National University of Mexico, Mexico City, Mexico, and the M.S. degree in computer science from Carnegie-Mellon University, Pittsburgh, PA, in 1976 and 1979, respectively.

Since 1975 he has been a Research Assistant in the Department of Computer Science, Institute for Applied Mathematics and Systems, National University of Mexico, and is currently on leave at Carnegie-Mellon University. He has been a graduate student at Carnegie-Mellon University since September 1976 and is currently finishing his Ph.D. His current research interests include software engineering, programming environments, and specifically, syntax-directed editing. He is presently working with the Gandalf Project in the design and implementation of a program development environment.

Mr. Medina-Mora is a student member of the Association for Computing Machinery.



Peter H. Feiler was born in Bad Toelz, Federal Republic of Germany, in 1952. He received the Vordiplom in mathematics and computer sciences from the Technical University in Munich in 1973.

Since September 1974 he has been a graduate student in computer sciences at Carnegie-Mellon University, Pittsburgh, PA, and is currently completing the Ph.D. degree. Since December 1980 he has been employed by Siemens Corporation with residence at Carnegie-Mellon University. He participated in the Family of Operating Systems project, in the design of STAROS, a multiprocessor operating system, and is currently involved in the design and implementation of a program development support environment (Gandalf Project). Other research interests include personal computing and local networks.

Mr. Feiler is a member of the Association for Computing Machinery.

An Experiment in Small-Scale Application Software Engineering

BARRY W. BOEHM

Abstract—**This paper reports the results of an experiment in applying large-scale software engineering procedures to small software projects. Two USC student teams developed a small, interactive application software product to the same specification, one using Fortran and one using Pascal. Several hypotheses were tested, and extensive experimenal data collected. The major conclusions were as follows.**

- **Large-project software engineering procedures can be cost-effectively tailored to small projects.**
- **The choice of programming language is not the dominant factor in small application software product development.**
- **Programming is not the dominant activity in small software product development.**
- **The "deadline effect" holds on small software projects, and can be used to help manage software development.**
- **Most of the code in a small application software product is devoted to "housekeeping."**

The paper presents the experimental data supporting these conclusions, and discusses their context and implications.

Index Terms—**Programming languages, programming methodology, software engineering, software management, software project data.**

Manuscript received April 18, 1980; revised December 29, 1980.

The author is with the Systems Engineering and Integration Division, TRW, Redondo Beach, CA 90278.

I. Introduction

Background

THE experiment described in this paper took place as part of a first-year graduate course in software engineering given at the University of Southern California (USC) in the Fall of 1978. It involved the development of a small (2000 deliverable source instructions) application software product: an interactive version of the COCOMO [1] model for estimating software costs. Two teams specified and developed independent versions of the same product, one team using Fortran and the other using Pascal.

The main reason for the project was to give the students experience in applying all the disciplines involved in practical software engineering: project planning, requirements specification, design, programming, testing, maintenance, management, technical communication, and human engineering of the man-machine interface. The choice of a cost estimation model as the product to be developed was based on three main criteria.

1) Its size appeared appropriate for the one-semester course schedule.

2) The subject matter was easy for students to understand.

0098-5589/81/0900-0482$00.75 © 1981 IEEE